



Estruturas de Decisão, Repetição e Funções

Tudo pronto para continuar? Nesta unidade vamos nos aprofundar mais na linguagem JavaScript, ela exigirá de você dedicação e atenção, pois é bastante coisa para aprender.

Então vamos ao que interessa!

Operadores de Comparação e Estruturas de Decisão

Muitas vezes, ao escrevermos nossos códigos, será necessário executar ações específicas e blocos de código diferentes, dado o estado atual de nossa aplicação. Por exemplo, se queremos verificar se algum jogador deve receber uma pontuação especial ou se algum personagem deve ser eliminado do jogo, pois perdeu todos os seus pontos de vida.

Faremos isso através das instruções condicionais e operadores de comparação, testando se alguma condição específica é verdadeira e, dessa forma, tomando decisões e caminhos diferentes dentro de nosso código.

Em JavaScript, podemos usar as seguintes instruções condicionais:

Instrução	Descrição
if	Executa um bloco de código se a condição especific... for **verdadeira**.
else	Usaremos esta instrução para executar um bloco de código se a condição for **falsa**.
else if	Usaremos **else if** para especificar novas condições, caso a primeira dada por **if** for falsa.
switch	Usada para executar blocos de códigos diferentes com base em condições diferentes.

Assim, a sintaxe básica para uma estrutura de decisão em JavaScript seria a seguinte:

```
if (condição) {
  // bloco de código a executar se a condição for verdadeira
}
```

Ou seja, usamos a instrução condicional **if** seguida da condição escrita entre parênteses **(condição)**. Já o bloco de código alternativo fica dentro das chaves **{ }**.

A condição geralmente será escrita com auxílio dos operadores de comparação. Eles são declarações lógicas que utilizamos para examinar a igualdade entre valores diversos, tipos de dados e variáveis.

Veja a seguir uma lista de alguns destes operadores:

Símbolo	Tipo da Condição	Significado
===	Igualdade estrita	Verifica se os valores da esquerda e direita são **iguais**.
!==	Estritamente não-igual	Compara se os valores da esquerda e direita **não são iguais**.
<	Menor que	Verifica se o valor da esquerda é **menor** que o da direita.
>	Maior que	Examina se o valor da esquerda é **maior** do que o valor da direita.
<=	Menor ou igual que	Verifica se o valor da esquerda é **menor ou igual** ao da direita.
>=	Maior ou igual que	Examina se o valor da esquerda é **maior ou igual** ao valor da direita.

Juntando as instruções condicionais com os operadores de comparação, temos o seguinte exemplo, usando a instrução **if** com o operador **menor que**:

```
if (ptosVida < 10) {
    aviso = "Tome cuidado!";
}
```

No exemplo acima, comparamos se os pontos de vida do jogador dado pela variável **ptosVida** estão menores do que o valor **10**. Caso isso aconteça, executaremos o bloco de código dentro de **{ }**, onde atribuímos um novo valor à variável **aviso**.

Já a sintaxe para a instrução **else**, em resumo, seria o seguinte:

```
if (condição) {
  // bloco de código a executar se a condição for verdadeira
} else {
  // bloco de código a executar se a condição for falsa
}
```

No caso da instrução **else if**, temos a seguinte sintaxe:

```
if (condição1) {
  // bloco de código a executar se a condição for verdadeira
```

```
} else if (condição2) {
  // código que será executado caso condução1 seja e
  // condição2 for verdadeira
} else {
  // bloco de código caso todas as condições sejam falsas
}
```

A linguagem JavaScript também contém o chamado **operador ternário**, que pode ser usado para atribuir um valor a uma variável com base em alguma condição. Por exemplo:

```
let aviso = (ptosVida < 10) ? "Tome cuidado!" : "Continue!";
```

Podemos ainda combinar comparações com **operadores lógicos**. Eles são usados para determinar a lógica entre duas ou mais condições. Por exemplo:

```
if (ptosVida < 10 && suprimento >= 1) {
  aviso = "Use seu suprimento para recuperar vida!";
}
```

Vejamos alguns desses operadores lógicos que podemos usar em nossas decisões em JavaScript:

Operador Lógico	Significado
&&	Combina duas condições, verificando se elas são estritamente verdadeiras.
||	Este operador testa duas condições e o teste passa bastando uma das condições ser verdadeira.
!	Testa se uma condição não é verdadeira.

Estruturas de Repetição

Em JavaScript é possível utilizar diferentes tipos de estruturas de repetição. Neste resumo vamos focar no mais comumente utilizados:

- **for**: usamos esta instrução quando queremos repetir um bloco de código dado um número finito de vezes.

- **while**: neste caso vamos repetir um bloco de código enquanto uma condição for verdadeira.

Para a sintaxe da instrução **for** devemos informar como ocorre o incremento de uma variável que consequentemente está ligada a uma condição de parada. Para isso, inserimos algumas instruções separadas por **;** dentro de parênteses. Por exemplo:

```
let i = 0;
```

```
let numAtaques = 3;
for (let i = 0; i < numAtaques; i++) {
    aviso = "Executando o ataque " + i;
}
```

Explicando:

- Estipulamos um limite de ataques dado por **let numAtaques = 3**;
- A primeira instrução dentro de **( )** declarar a variável de incremento, juntamente com seu valor inicial. A variável então será **i** inicializada em **0**;
- Em seguida, definimos a condição que manterá a repetição. Nesse caso, o loop se manterá enquanto **i** for menor que **numAtaques**;
- Na terceira instrução, definimos o fator de incremento de **i**. Nesse caso, será um incremento unitário, pois **i++**.

Então, a cada vez que o bloco de código dentro de **{ }** for executado, **i** será incrementado em uma unidade. Isso irá ocorrer enquanto **i** for menor que **numAtaques**, isto é, três vezes, já que **numAtaques = 3**.

Para o caso da estrutura de repetição **while**, sua sintaxe é mais simples. Devemos indicar a condição dentro de **( )** que será verificada a cada execução do bloco na declaração da instrução. Por exemplo:

```
let numAtaques = 3;
while (i < numAtaques) {
    aviso = "Executando o ataque " + i;
    i++;
}
```

Este é o mesmo exemplo que foi mostrado anteriormente, mas usando **while**. Perceba que devemos sempre inserir a instrução de incremento da variável **i**. Isso não precisa ocorrer necessariamente por último, como no exemplo. Mas, se você esquecer desse incremento, o loop nunca terminará e ocorrerá um loop eterno, que consequentemente irá travar o navegador.

Trabalhando com Vetores

Um **array** é tipo de dados especial em qualquer linguagem de programação. Podemos dizer que é um tipo de variável que nos permite armazenar mais de um valor ou conjunto de valores dentro da mesma variável. Dessa forma, acessaremos os valores contidos neste **array** através de um número de índice.

Por exemplo, podemos armazenar nossa coleção de *pokémons* no nosso *pokedex* pessoal. Poderíamos declarar a variável **pokedex** da seguinte forma:

```
const pokedex = new Array();
```

Para adicionar *pokémons* neste inventário, usamos as seguintes instruções, referenciando sempre o índice onde ele deve ser armazenado dentro de **[ ]**:

```
pokedex[0] = "Abra";
pokedex[1] = "Dragonite";
pokedex[2] = "Porygon";
pokedex[3] = "Alakasam";
```

Ou seja, este é um excelente recurso para armazenar dados em inventários do jogador em nossos jogos! Mas observe com cuidado: o primeiro elemento de um vetor em JavaScript será sempre no índice **0** (zero).

Podemos combinar o uso de vetores com as estruturas de repetição. No exemplo a seguir, vamos percorrer o vetor **pokedex** exibindo uma mensagem no console para cada *pokémon* armazenado:

```
for (let i = 0; i < pokedex.length; i++) {
    console.log(pokedex[i] + " eu escolho você!");
}
```

Neste exemplo, quando o valor de **i** for igual a **2**, a mensagem será **"Porygon eu escolho você!"**.

Perceba ainda que utilizamos a propriedade **length** padrão de vetores JavaScript. Nesse caso, queremos dizer que vamos executar o bloco de código até atingir o total de valores armazenados. No caso do vetor **pokedex**, que fizemos como exemplo, esta propriedade terá o valor de **pokedex.length** igual a 4, pois temos 4 *pokémons* armazenados.

Utilizando Funções

Uma função JavaScript é um bloco de código projetado para executar tarefas específicas ou repetitivas e que serão invocadas pelo programador dentro de seu código fonte quando for necessário.

Uma função JavaScript é definida com a palavra-chave **function**, seguida por um nome mais parênteses **( )**. O bloco de código a ser executado pela função será inserido entre chaves **{ }**. Veja a sintaxe:

```
function nomeFuncao() {
  // bloco de código a ser executado pela função
}
// invocar a função
nomeFuncao();
```

Devemos declarar nossas funções usando as mesmas regras das variáveis: usaremos letras, dígitos, sublinhados e cifrões. Caracteres especiais da língua portuguesa não são permitidos como o cedilha e caracteres acentuados, por exemplo.

No entanto, somente declarar a função não é suficiente para que ela execute o bloco de código contido nela. Uma função JavaScript só é executada quando a invocamos em algum momento dentro de nosso programa. Para chamar a função, só precisamos fazer a referência para o nome dela em nosso código.

Os parênteses podem incluir parâmetros separados por vírgulas para que sejam calculados e utilizados dentro da função. Esses são os argumentos da função e irão se comportar dentro do escopo da mesma como variáveis locais. A sintaxe seria como a seguir:

```
function nomeFuncao(param1, param2) {
  // bloco de código a ser executado pela função
  // cálculos usando os parâmetros param1 e param2
}
```

Para nomear os parâmetros de uma função, devemos tomar os cuidados e usar as mesmas boas práticas de quando nomeamos nossas variáveis. Além disso, por se tratarem de variáveis locais, os parâmetros serão variáveis acessíveis somente dentro da função.

Em JavaScript as funções também podem retornar valores diversos como resultado. Isso será muito comum e também muito útil para os vários propósitos que iremos nos deparar ao projetar nossos jogos. Usaremos para isso a palavra reservada **return** dentro da função e essa será a última instrução executada no seu escopo.

Veja um exemplo:

```
// Função que calcula o dano de um ataque
// Parâmetro ptsAtaque: pontos de ataque do atacante
// Parâmetro ptsDefesa: pontos de defesa do defensor
function danoDeAtaque(ptsAtaque, ptsDefesa) {
    // Calcula a quantidade de dano e retorna o resultado
  return ptsAtaque - ptsDefesa;
}
```

```
// Pontos de ataque do Charmander
let ptosAtqCharmander = 40;
// Pontos de defesa do Bulbasaur
let ptosDfsBulbasaur = 30;
// Charmander ataca Bulbasaur, qual seria o dano?
let dano = danoDeAtaque(ptosAtqCharmander, ptosDfsBulbasaur);
```

Chegamos ao final de mais um resumo, este foi cheio não? Mas até aqui já aprendemos muita coisa para um conhecimento intermediário em JavaScript. Na próxima unidade, veremos como conectar tudo isso que aprendemos com nossos documentos HTML, para acessar seus elementos e também o CSS.

Abraços e te vejo lá!

Atividade Extra



Em JavaScript são muitos os métodos e recursos úteis para trabalharmos com vetores. Como atividade extra, acesse o site da W3 Schools abaixo, leia o conteúdo e experimente os vários exemplos contidos lá, principalmente para os métodos **pop()** e **push()**, que você terá de usar muito na sua vida de programador web. Pra realizar os experimentos clique no botão verde com a mensagem ***Try it Yourself*** (tente você mesmo).

- **JavaScript Array Methods**
  Disponível em: <https://www.w3schools.com/js/js_array_methods.asp>.

Referência Bibliográfica

MDN WEB DOCS. **Elementos construtivos do Javascript.** Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Building_blocks>.

W3 Schools. **JavaScript Tutorial.** Disponível em: <https://www.w3schools.com/js/default.asp>.

Ir para exercício